

SPECIFICATION

IBM Docket No. ST9-95-010

Express Mail #: EG340732293US

TO ALL WHOM IT MAY CONCERN:

BE IT KNOWN that We, Dz Ching Ju of Sunnyvale, California and citizen of the Taiwan, Kalyan Muthukumar of Cupertino, California and citizen of India, Shankar Ramaswamy of Bethel Park, Pennsylvania, and citizen of India, and Barbara Bluestein Simons of Palo Alto, California, and citizen of the United States, have invented new and useful improvements in

A METHOD OF, SYSTEM FOR, AND COMPUTER PROGRAM PRODUCT FOR PROVIDING EFFICIENT UTILIZATION OF MEMORY HIERARCHY THROUGH CODE RESTRUCTURING

of which the following is a specification:

1 **TITLE OF THE INVENTION**

2
3 **A METHOD OF, SYSTEM FOR, AND COMPUTER PROGRAM PRODUCT FOR**
4 **PROVIDING EFFICIENT UTILIZATION OF MEMORY HIERARCHY**
5 **THROUGH CODE RESTRUCTURING**
6
7
8

9
10
11 **CROSS REFERENCES TO RELATED APPLICATIONS**
12
13
14
15
16

17 **STATEMENT AS TO RIGHTS TO INVENTIONS MADE UNDER FEDERALLY**
18 **SPONSORED RESEARCH AND DEVELOPMENT**
19
20
21

22
23
24 A portion of the Disclosure of this patent document contains material which is subject to
25 copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone
26 of the patent document or the patent disclosure, as it appears in the Patent and Trademark Office
27 patent file or records, but otherwise reserves all copyright rights whatsoever.

1 BACKGROUND OF THE INVENTION

2
3 *Ins a17*
4 1. Field of the Invention

5
6 This invention relates generally to optimizing compilers for development of computer
7 programs for use on a computer, and more particularly to code restructuring.

8
9
10 2. Description of the Related Art

11
12 Advances in processor designs have provided very high speed instruction processing. In order
13 to sustain this speed, processors need very effective usage of the memory hierarchy of the machines.
14 Ensuring effective usage is difficult for large programs like compilers or database engine codes that
15 have complex control flow. This occurs, for example, as a result of having to consider inputs of
16 different types or handle error conditions appropriately. Consequently, code generated for large
17 complex programs consists of sections that are heavily executed intermingled with sections that are
18 infrequently executed, resulting in poor utilization of the memory hierarchy. *Code restructuring* may
19 improve memory hierarchy utilization and, consequently, performance.

20
21 The state of the art of code restructuring techniques is described in Simons (B. Simons, "Code
22 Positioning for Procedures and Basic Blocks," IBM Technical Report ADTI-1994-006; also available
23 as IBM Santa Teresa Laboratory Technical Report TR 03.580.). Two representative techniques
24 described by Simons include the IBM Heatshrink and the IBM AOPT products. Heatshrink, a trace-
25 directed program restructuring (TDPR) tool, was designed and implemented by Randy R. Heisch. A
26 more detailed description of Heatshrink is given in Heisch (R. R. Heisch, "Trace-Directed Program
27 Restructuring for AIX Executables", *IBM Journal of Research and Development*, vol. 38, September
28 1994.). AOPT, a stand-alone tool developed by Itai Nahshon, attempts to locate a "hot path" in the
29 code and then move the entire path as a single entity, rather than move individual basic blocks.

1 Profile information, used by both Heatshrink and AOPT, is obtained by AOPT. Profiling involves
2 running code using sample or representative inputs. Based on the behavior of the code on the given
3 inputs, one makes assumption about its behavior on all inputs. Heatshrink and AOPT both perform
4 global reordering of basic blocks and are bundled together as part of FDPR (Feedback Directed
5 Program Restructuring tool), an IBM Licensed Program Product for IBM AIX Operating System,
6 Version 4.1.

7
8 Pettis et al. (K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," *Proceedings of*
9 *the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (PLDI)*,
10 pp. 16-27, June 1990.) follow a different approach, that also relies on profiling information, but that
11 moves executed basic blocks only *within* a procedure in addition to moving procedures. Basic blocks
12 that are not executed according to the profile data are moved to a separate region at the end of an
13 ordering.

14
15 These and other conventional techniques for code restructuring attempt to keep frequently
16 executed code close together in memory. Intuitively, such an approach tries to retain the working set
17 of frequently executed code in cache. If the working set fits in the cache, this approach is reasonable;
18 on the other hand, if the cache is small or if the effective portion of the cache available to an
19 application is small due to multiprogramming, conventional code restructuring techniques may not
20 perform well. A major flaw of these conventional approaches is that "closeness" in memory does not
21 actually relate to cache or paging performance. Two basic blocks may be "close" together in
22 memory, and yet there may still be a cache miss or page fault transitioning between the two basic
23 blocks, or they may be very far apart and still cause the same effect of a cache miss or page fault.
24 Thus, conventional code restructuring techniques fail to realize that blocks interacting frequently
25 should be kept in the same cache line or page rather than "close" to each other in memory.
26 Conventional code restructuring techniques also fail to realize that memory hierarchy hardware
27 parameters such as cache line size, page size, cache associativity size and translation lookaside buffer
28 (TLB) size may be considered in determining the code structure of an application.
29

SUMMARY OF THE INVENTION

The invention disclosed herein comprises a method of, a system for, and an article of manufacture for providing code restructuring based on profiling information and memory hierarchy, that is also scalable to arbitrary levels of memory hierarchy.

The present invention reorders code to reduce communications across cache line and across page boundaries. The present invention also reorders code to reduce the likelihood that chunks of code that communicate with each other and that have the same cache or TLB associativity set will be assigned to different cache lines or pages. The present invention may be generalized to deal with multiple levels of cache or other types of memory hierarchy.

The present invention accomplishes this by reordering code such that chunks of code that communicate extensively with each other are on the same cache line, page, or other level of memory hierarchy. Reordering is performed by first clustering the nodes, performing first level padding of unfilled clusters with basic blocks that were not accessed in profiling, and then performing additional padding using no-operation instructions (NOP or NO-OP) to fill out any unfilled portion of that level of the memory hierarchy. For example, to reorder code for a first level cache line, nodes are clustered to reduce communications across cache lines, any of these clusters which are unfilled clusters are padded with basic blocks that were not accessed in profiling, and then any unfilled cache lines are further padded with NOPs to fill out the cache lines.

The present invention avoids set collisions by considering both memory hierarchy size and associativity. Associativity means that there is a connection between a memory location and the place in the memory hierarchy to which the data for that memory location is assigned. 4-way associativity, for example, means that the cache is divided into four sets, and each memory location is uniquely mapped into one of those four sets. This mapping may be done by using two bits of the memory location address. If a cache line is 128 bytes in size, and if a memory word contains 4 bytes, then 32 words will fit on one cache line. A primary objective of the associativity mapping is to have each

consecutive chunk of 32 words map to different associativity sets. If bits 6 and 7 of the memory location address are used to determine to which associativity set a 32 bit chunk maps, then adjacent 32 word chunks map to different sets. A set collision occurs when all the cache lines in a particular set are full. This may occur even if one of the other sets has space. As conventional processor implementations hard wire the associativity between memory and cache sets, data may not arbitrarily occupy a cache line in an associativity set that differs from the one to which the particular memory is assigned. The present invention considers both memory hierarchy size and associativity to avoid such associativity set collisions by dividing the cache size by its associativity, i.e., if an 8K cache is 4-way set associative, then $n_2 = 2048$ bytes. This n_2 falls between the cache line size, n_1 , of 128 bytes and the page size, n_3 , of 4096 bytes. Similarly, the TLB size divided by its associativity is considered. Alternative embodiments of the present invention may be modified to ignore various levels of the memory hierarchy; for example, an alternative embodiment may ignore associativity.

The present invention restructures code by constructing a *Program Execution Graph* (PEG) corresponding to a level of the memory hierarchy, partitioning this PEG to reduce estimated memory overhead costs, and constructing a PEG for a next level of the memory hierarchy from the partitioned PEG. The present invention may be performed by:

- a. constructing a *Program Execution Graph* (PEG) from control flow and frequency information from a profile of the program to be restructured. The profile of the program is produced by running the program using sample or representative inputs. The present invention assumes that the profiled program behavior is representative of the general program behavior, i.e., that basic blocks that communicate in the profiling will communicate in general. This type of profiling assumption is commonly used in practice by those skilled in the art. The PEG is a weighted undirected graph comprising nodes representing basic blocks and edges representing transfer of control between pairs of basic blocks. The weight of a node is the number of bytes in the basic block it represents, and the weight of an edge is the frequency of transition in the profiling between the pair of basic blocks it connects. Those skilled in the art recognize that the unit of measurement for the weight may be a unit other than bytes, i. e., bits, words, etc.

b. partitioning or clustering the nodes of the PEG into clusters such that the sum of the weights of the nodes in any cluster is no greater than an upper bound, and the sum of the weights of the edges between the clusters is minimized.

c. constructing a next PEG from the clusters of the partitioned PEG such that a node in the next PEG corresponds to a cluster in the partitioned PEG, and such that there is an edge between two nodes in the next PEG if there is an edge between the clusters represented by the two nodes.

d. assigning weights to the nodes and edges of the next PEG to produce a PEG.

- e. repeating steps b through e for each level of the memory hierarchy.

f. restructuring the basic blocks corresponding to the clusters into contiguous code starting at an appropriate memory boundary, and adding NOPs as needed so that the number of memory locations occupied by each cluster is equal to an upper bound of the cluster weight.

In accordance with one aspect of this invention, a Program Execution Graph constructor constructs a Program Execution Graph (PEG), a weighted undirected graph comprising nodes representing basic blocks or clusters of basic blocks and edges representing transfer of control between pairs of basic blocks where each basic block in the pair is in a different cluster, wherein a weight of a node of the PEG is a size of a represented basic block, and a weight of an edge is a frequency of transition between a pair of basic blocks that the edge connects or a frequency of transition between elements of two clusters that the edge connects.

In accordance with another aspect of this invention, the PEG, a weighted undirected graph $G = (V, E)$ with edge and node weights, is partitioned into clusters such that the sum of the weights of the edges whose endpoints are in different partitions is minimized and the sum of weights of nodes in each set is no greater than a given upper bound.

1 In accordance with another aspect of this invention, a PEG for level i of a memory hierarchy,
2 $G_i = (V_i, E_i)$, is partitioned into clusters, and these clusters from level i are the nodes for $G_{i+1} = (V_{i+1},$
3 $E_{i+1})$, a PEG for level $i+1$ of the memory hierarchy.

4
5 In accordance with another aspect of this invention, if u and $v \in V_{i+1}$, there is edge (u, v)
6 between u and v if and only if there is an edge between some pair of nodes in G_i , one of which is in
7 the cluster corresponding to u and one of which is in cluster corresponding to v .

8
9 In accordance with another aspect of this invention, the weight of edge (u, v) in G_{i+1} is the sum
10 of the weights of the edges between component nodes of u and v in G_i .

11
12 In accordance with another aspect of this invention, the upper bound for a level of the
13 memory hierarchy may be equal to the size of that level of the memory hierarchy.

14
15 In accordance with another aspect of this invention, the upper bound for a level of the
16 memory hierarchy may be equal to a multiple of the size of that level of the memory hierarchy.

17
18 The present invention has the advantage of restructuring code responsive to a memory
19 hierarchy.

20
21 The present invention has the further advantage of reducing memory hierarchy access cost of
22 restructured code.

23
24 The present invention has the further advantage of reducing cache misses between basic
25 blocks of restructured code.

26
27 The present invention has the further advantage of reducing page faults between basic blocks
28 of restructured code.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention and the advantages thereof, reference is now made to the Detailed Description in conjunction with the attached Drawings, in which:

Figure 1 shows a Program Execution Graph G_1 of a first memory hierarchy level;

Figure 2 shows a partitioned Program Execution Graph G_1 produced in accordance with the present invention;

Figure 3 shows a Program Execution Graph G_2 of a second memory hierarchy level produced from the clusters of the partitioned Program Execution Graph G_1 in accordance with the present invention;

Figure 4 shows a partitioned Program Execution Graph G_2 produced in accordance with the present invention;

Figure 5 shows a Program Execution Graph G_3 of a third memory hierarchy level produced from the clusters of the partitioned Program Execution Graph G_2 in accordance with the present invention;

Figure 6 and **Figure 7** are flowcharts illustrating the operations preferred in carrying out the present invention; and

Figure 8 shows a computer system used in performing the method of the present invention, forming part of the apparatus of the present invention, and which may use the article of manufacture comprising a computer-readable storage medium having a computer program embodied in said medium which causes the computer system to practice the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring now to Figure 1 through Figure 5, sample Program Execution Graphs are shown to illustrate the practice of the present invention. The control flow and frequency information of the profiled program is organized in the form of a Program Execution Graph (PEG) which is a weighted undirected graph. For example, Figure 1 illustrates a PEG G_i , generally referred to as 101, where $G_i = (V_i, E_i)$. A node in the set of nodes V_i corresponds to a basic block that has been executed at least once during profiling. For G_i 101, the set of nodes V_i comprises the set {A 103, B 105, C 107, D 109, E 111, F 113, G 115, H 117, I 119, J 121, K 123, L 125, M 127, N 129, O 131, P 133, and Q 135}. An edge (u,v) , in the set of edges E_i , between nodes u and v implies that there is some transition that occurs during profiling between the two basic blocks represented by u and v . For G_i 100, the set of nodes E_i comprises the set {(A, B) 137, (B, C) 139, (C, F) 141, (F, K) 143, (K, L) 145, (L, N) 147, (N, O) 149, (O, P) 151, (O, Q) 153, (B, N) 155, (F, J) 157, (J, K) 159, (B, E) 161, (E, F) 163, (B, D) 165, (D, E) 167, (E, G) 169, (G, I) 171, (I, L) 173, (G, H) 175, (H, I) 177, (H, M) 179, (M, N) 181, and (D, M) 183}. The weight of a node v , written $W_i(v)$, is the size (in bytes) of the basic block that v represents. In G_i 101, node A 103 has a weight $W_i(A)$ of 114 bytes, node B 105 has a weight $W_i(B)$ of 36 bytes, C 107 has a weight $W_i(C)$ of 8 bytes, and so on. The weight of edge $e = (u,v)$, written $W_i(e)$, is the total number of transitions from basic block u to basic block v plus the total number of transitions from basic block v to basic block u . In G_i 101, edge (A, B) 137 has a weight $W_i((A, B))$ of 556 transitions, edge (B,C) 139 has a weight $W_i((B,C))$ of 745 transitions, edge (C, F) 141 has a weight $W_i((C, F))$ of 745 transitions, and so on.

Using the initial PEG G_i for a program and given all the parameters for a n -level memory hierarchy, the present invention performs code restructuring in n or fewer steps. The i^{th} step constructs a new PEG G_{i+1} by clustering PEG G_i wherein graph partitioning is performed on G_i to produce the clustering. Clustering and graph partitioning are discussed in the following paragraphs.

1 A cluster, written Cl , is defined to be a set of nodes. A clustering of a graph G with upper
2 bound N , written $Clust(G, N)$, is a partitioning of the nodes in G into clusters so that the sum of the
3 node weights of the nodes in any cluster is no greater than N . Each cluster in a partitioned PEG
4 represents a unit in the level of the hierarchy for which clustering is performed. For example, in the
5 first stage, a unit represents the size of a cache line (in bytes) or some integral multiple of this size (to
6 allow for basic blocks that are larger than a cache line).

7
8 The goal of a graph partitioning heuristic is to create clusters of nodes from the PEG such that
9 the sum of the weights of the edges that have endpoints in different clusters is minimized, subject to
10 the total weight of the nodes in each cluster not exceeding N . This goal is modeled by using a cost
11 function $C()$. For edge e , e has zero cost in $Clust(G, N)$, written $C(e) = 0$, if both endpoints of e are in
12 the same cluster for $Clust(G, N)$. Otherwise, e has a cost in $Clust(G, N)$, $C(e) = 1$. Those skilled in the
13 art recognize that costs other than zero or one may be assigned by the cost function for the purposes
14 of partitioning the graph. The goal is to compute a clustering that minimizes the sum of $C(e)$ times
15 $W(e)$ for all edges e in E . The result of graph partitioning is to create the clusters, i.e., to assign
16 values to $C(e)$ for all edges e in E . If $C(e) = 0$, then the endpoints of edge e are in the same cluster;
17 otherwise, the endpoints of edge e are in different clusters.

18
19 Each cluster represents a unit in the stage of the hierarchy for which clustering is performed.
20 Because a basic block may be larger than a cache line, the first stage of the heuristic allows a cluster
21 to be the size of some multiple of the number of bytes in a cache line. At each subsequent stage, the
22 clusters are limited to the size of the portion of the memory hierarchy being considered, for example,
23 a memory page size. However, if for some reason this cluster size is not preferred, the heuristic may
24 be easily modified to assign a different value to the cluster size.

25
26 The cost function is an approximation of the memory hierarchy overhead incurred by the
27 restructured program on a real machine. Formally, the code restructuring problem may be defined as
28 a first minimization of Φ_1 , then a minimization of Φ_2 , then a minimization of Φ_3, \dots , and a final
29 minimization of Φ_k , where k is the number levels in the memory hierarchy, and where:

$$\Phi_i = \sum_{e \in E_i} C(e) \times W(e)$$

The objective is to construct a clustering $Clust(G, N_i)$ so that Φ_i is the summation of the product of $C(e)$ and $W(e)$ for all e in E_i ; where $C(e) = 0$ if both endpoints of e are in the same cluster and $C(e) = 1$ otherwise; where the sum of the node weights of the nodes in each cluster in $Clust(G, N_i)$ is no greater than N_i , and where N_i is some multiple of the cache line size.

For example, if a cache line size is 128 bytes, N_i may be set to 128 bytes, and G_i may be partitioned into clusters such that the sum of the node weights of the nodes in each cluster is no greater than 128 bytes where N_i is a multiple of one times the cache line size. Such a partitioning of G_i yields a clustering $Clust(G_i, N_i) = \{\{A\ 103\}, \{B\ 105, C\ 107, F\ 113\}, \{J\ 121\}, \{K\ 123, L\ 125\}, \{D\ 109, E\ 111\}, \{G\ 115, H\ 117, I\ 119\}, \{M\ 127, N\ 129\}, \{O\ 131, P\ 133, Q\ 135\}\}$. Clustering $Clust(G_i, N_i)$ is illustrated in Figure 2 where cluster 205 comprising $\{A\ 103\}$ has a node weight sum of 114 bytes; cluster 210 comprising $\{B\ 105, C\ 107, F\ 113\}$ has a node weight sum of 126 bytes; cluster 215 comprising $\{J\ 121\}$ has a node weight sum of 92 bytes; cluster 220 comprising $\{K\ 123, L\ 125\}$ has a node weight sum of 94 bytes; cluster 225 comprising $\{D\ 109, E\ 111\}$ has a node weight sum of 86 bytes; cluster 230 comprising $\{G\ 115, H\ 117, I\ 119\}$ has a node weight sum of 120 bytes; cluster 235 comprising $\{M\ 127, N\ 129\}$ has a node weight sum of 80 bytes; and cluster 240 comprising $\{O\ 131, P\ 133, Q\ 135\}$ has a node weight sum of 104 bytes.

Once a clustering on G_i is constructed, "padding" is performed by adding basic blocks that were not executed during the profiling to clusters that have capacity, i.e. where the sum of the weights of the nodes in the cluster is less than N_i , and where adding the un-executed basic block will not cause the sum of the cluster's weights to exceed N_i . If there are still basic blocks remaining that have not been added to any of the clusters, and if there is no more cluster capacity to absorb these basic blocks, then they are grouped into additional clusters of their own. The technique of adding the un-executed basic block nodes to the clusters or of putting them into their own clusters is called bin packing. A greedy bin packing heuristic may be used for performing the bin packing.

Although the example of **Figure 2** does not contain any un-executed basic blocks that may be used for padding (as the initial PEG does not include un-executed basic blocks in its node set), if there is an additional unexecuted basic block whose size is less than 48 bytes, it may be padded into the cluster 235 with node M 127 and node N 129 yielding a padded cluster whose sum of node weights does not exceed $N_i = 128$. In a similar manner, a greedy bin packing heuristic may pad the other clusters.

Once all of the basic blocks have been put into some cluster, then the next stage is performed which is the construction of a graph $G_2 = (V_2, E_2)$. Each node in V_2 corresponds to a cluster in G_1 , and each node in V_2 has a weight of one. This weight corresponds to the assignment of N_i bytes to the basic blocks in each cluster. An edge $e = (u, v)$ exists in E_2 if and only if both u and v are in V_2 , u and v correspond to different clusters, and there is some edge in E_1 from one of the nodes in the cluster represented by u to some node in the cluster represented by v , or visa versa. The weight of e is the sum of the weights of all edges in E_1 that have endpoints in the two clusters (u and v).

Referring next to **Figure 3**, a graph $G_2 = (V_2, E_2)$ constructed from the clusters $Clust(G_1, N_i)$ of G_1 is illustrated. G_2 is constructed by creating a node corresponding to each cluster of $Clust(G_1, N_i)$ such that node A 305 corresponds to cluster 205, node BCF 310 corresponds to cluster 210, node J 315 corresponds to cluster 215, node KL 320 corresponds to cluster 220, node DE 325 corresponds to cluster 225, node GHI 330 corresponds to cluster 230, node MN 335 corresponds to cluster 235, and node OPQ 340 corresponds to cluster 240. Each of the nodes $V_2 = \{A\ 305, BCF\ 310, J\ 315, KL\ 320, DE\ 325, GHI\ 330, MN\ 335, OPQ\ 340\}$ is assigned a weight of one. Edges E_2 are then constructed between various nodes V_2 where an edge $e = (u, v)$ exists in E_2 if and only if both u and v are in V_2 , u and v correspond to different clusters, and there is some edge in E_1 from one of the nodes in the cluster represented by u to some node in the cluster represented by v , or visa versa. For example, edge (BCF, DE) 350 is created as there is an edge (B, D) 165 between node B 105 and node D 109, and as node B 105 and node D 109 are in different clusters, node B 105 being in cluster 210 and node D 109 being in cluster 225. In a similar manner, edges (A, BCF) 345, (BCF, KL) 355, (BCF, J) 365, (BCF, MN) 370, (DE, MN) 375, (DE, GHI) 380, (GHI, MN) 385, (GHI, KL) 390, (J,

KL) 392, (KL, MN) 394, and (MN, OPQ) 396 are created. Weights are then assigned to the edges of E_2 where the weight of an edge (u, v) in E_2 is the sum of the weights of all edges in E_1 that have endpoints in the two clusters u and v . For example, edge (BCF, DE) 350 is assigned a weight of 440, the sum of the edge weights of edge (B, D) 165, edge (B, E) 161, and edge (E, F) 163, all of these edges having endpoints in cluster 210 and cluster 225. In a similar manner, edge weights of 556, 700, 750, 88, 200, 360, 800, 112, 128, 470, and 500, respectively, are assigned to edges (A, BCF) 345, (BCF, KL) 355, (BCF, J) 365, (BCF, MN) 370, (DE, MN) 375, (DE, GHI) 380, (GHI, MN) 385, (GHI, KL) 390, (J, KL) 392, (KL, MN) 394, and (MN, OPQ) 396.

Let n_2 be the size of the next level of the memory hierarchy, typically the page size in bytes. Set $N_2 = n_2/N_1$. Then the same clustering heuristic is applied to G_2 , using the same cost function C , i.e. counting only those edges that have end points in different clusters in G_2 . This partitioning of G_2 is used to construct graph $G_3 = (N_3, E_3)$ by making each node in V_3 correspond to a cluster in V_2 , and having an edge in E_3 if and only if there is some edge in E_2 that has end points in nodes in the two clusters represented by the endpoints of the E_3 edge. Again the nodes have a weight of one, and the edges have weights corresponding to the sum of the weights on the edges in G_2 that run between some node in cluster u and some node in cluster v .

For example, if n_2 , the size of the next level of the memory hierarchy, is 512 bytes, then set $N_2 = n_2/N_1 = 512/128 = 4$. G_2 may be partitioned into clusters such that the sum of the node weights of the nodes in each cluster is no greater than 4. Such a partitioning of G_2 yields a clustering $Clust(G_2, N_2) = \{ABCFJKL\ 410 \text{ and } DEGHIMNOPQ\ 420\}$ as illustrated in Figure 4. A graph $G_3 = (V_3, E_3)$ may be constructed from the clusters $Clust(G_2, N_2)$ of G_2 as illustrated in Figure 5. G_3 is constructed by creating a node corresponding to each cluster of $Clust(G_2, N_2)$ such that node ABCFJKL 510 corresponds to cluster 410, and node DEGHIMNOPQ 520 corresponds to cluster 420. Each of the nodes $V_3 = \{ABCFJKL\ 510, DEGHIMNOPQ\ 520\}$ is assigned a weight of one. Edges E_3 are then constructed between various nodes V_3 where an edge $e = (u, v)$ exists in E_3 if and only if both u and v are in V_3 , u and v correspond to different clusters, and there is some edge in E_2 from one of the nodes in the cluster represented by u to some node in the cluster represented by v , or

visa versa. For example, edge (ABCFJKL, DEGHIMNOPQ) 530 is created as there is an edge (BCF, DE) 350 between node BCF 310 and node DE 325 which are in different clusters, node BCF 310 being in cluster 410 and node DE 325 being in cluster 420. Weights are then assigned to the edges of E_3 , where the weight of an edge (u, v) in E_3 is the sum of the weights of all edges in E_2 that have endpoints in the two clusters u and v . For example, edge (ABCFJKL, DEGHIMNOPQ) 530 is assigned a weight of 1110, the sum of the edge weights of edge (BCF, DE) 350, edge (KL, GHI) 390, edge (KL, MN) 470, and edge (BCF, MN) 370, all of these edges having endpoints in cluster 510 and cluster 520.

This process of graph partitioning may be repeated for each level of the memory hierarchy for $i > 1$. For level i of the memory hierarchy, let n_i be the size of the memory hierarchy, and set $N_i = n_i/N_{i-1}$. Then the same clustering heuristic is applied to G_n using the same cost function C , i.e. counting only those edges that have end points in different clusters in G_i . This partitioning of G_i is used to construct graph $G_{i+1} = (N_{i+1}, E_{i+1})$ by making each node in V_{i+1} correspond to a cluster in V_n and having an edge in E_{i+1} if and only if there is some edge in E_i that has end points in nodes in the two clusters represented by the endpoints of the E_{i+1} edge. Again the nodes have a weight of one, and the edges have weights corresponding to the sum of the weights on the edges in G_i that run between some node in cluster u and some node in cluster v .

One of the advantages of the present invention is that it may be generalized to deal with memory hierarchies with an arbitrary number of levels. In particular, the present invention may be applied to future improvements to memory hierarchy hardware.

Another advantage of the present invention is that it may be modified to deal with special cases. An example of such a special case occurs when a pair of basic blocks need to be close in memory. In this case, the edge connecting the two basic blocks may be given an exceedingly large weight, thereby forcing the two basic blocks into the same cluster if together they don't exceed the upper bound on the cluster size. This approach may also be generalized to the case in which several basic blocks are required to be close together in memory.

1 The present invention may also be modified to deal with the case of a program with a few very
2 large basic blocks. Assume basic blocks B_i , B_j , and B_k have weights larger than the size of the cache
3 line, and that the weights of all the other basic blocks are no greater than the cache line size. Define
4 $G'_1 = G_1 - \{B_i, B_j, B_k\}$, i.e. the PEG that is obtained by removing B_i , B_j , and B_k from the problem input.
5 Let N_1 = the cache line size, and let $Clust(G'_1, N_1)$ be the clustering that is computed by the heuristic.
6 $Clust(G'_1, N_1) + \{B_i, B_j, B_k\}$ may then be used as the input node set to the next stage of the heuristic.

7
8 As the nodes representing B_i , B_j , and B_k have weights greater than the value used for N_1 in the
9 first stage of the partitioning, the weight of the node in V_2 that corresponds to B_i is p_i , where p_i is the
10 smallest integer such that $p_i * N_1 \geq$ the number of bytes in B_i . Values p_j and p_k for B_j and B_k ,
11 respectively, are similarly computed. The heuristic may then be applied to G_2 , with the above
12 modifications. Note that this assumes that none of B_i , B_j , and B_k is larger in bytes than n_2 . If that is
13 not the case, then again the very large basic block(s) may be removed from consideration and
14 reintegrated at the next stage of the heuristic.

15
16 Referring next to **Figure 6** and **Figure 7**, flowcharts illustrating operations preferred in
17 carrying out the present invention are shown. In the flowcharts, the graphical conventions of a
18 diamond for a test or decision and a rectangle for a process or function are used. These conventions
19 are well understood by those skilled in the art, and the flowcharts are sufficient to enable one of
20 ordinary skill in the art to write code in any suitable computer programming language.

21
22 Referring now to **Figure 6**, the process of the invention, generally referred to as **600**, begins
23 at process block **610**. Thereafter, process block **615** determines memory hierarchy parameters such
24 as the number of levels in the memory hierarchy and the size of each such level, and process block
25 **620** profiles the target program. The profiler gathers information comprising identification of basic
26 blocks, control flow between basic blocks, and frequencies of transition between basic blocks. The
27 present invention assumes a program is divided into basic blocks, where a basic block is a contiguous
28 sequence of instructions with a single entry into and a single exit out of this sequence. In addition,
29 the present invention assumes information about the transfer of control between basic blocks and the

frequency with which the transfer occurs during actual program execution. This information may be made available through the use of profiling techniques well known to those skilled in the art such as the teachings summarized in Simons (B. Simons, "Code Positioning for Procedures and Basic Blocks," IBM Technical Report ADTI-1994-006; also available as IBM Santa Teresa Laboratory Technical Report TR 03.580.). Responsive to the profiler provided information, process block 625, a Program Execution Graph constructor, constructs a Program Execution Graph (PEG), a weighted undirected graph comprising nodes representing basic blocks and edges representing transfer of control between pairs of basic blocks. In this PEG produced by the Program Execution Graph constructor, a weight of a node of the PEG is a size of a represented basic block, and a weight of an edge is a frequency of transition in the profiling between a pair of basic blocks that the edge connects. Process block 630 then initializes a memory hierarchy level i to one, and sets n_i equal to a smallest power of 2 that is a multiple of the cache line size and that is greater than the sizes of all basic block sizes. Thereafter, process block 635 applies a heuristic H to partition G_i into $Clust(G_i, N_i)$, a partitioning of the nodes in G_i into clusters so that the sum of the node weights of the nodes in any cluster is no greater than N_i . The heuristic H solves the relaxed graph partitioning problem, and such heuristics are well known to those skilled in art such as the teachings of Barnard et al. (S. T. Barnard and H. D. Simon, "Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems," *Concurrency: Practice and Experience*, vol.6(2), pp.101-117, April 1994.). After clustering is performed on G_i , process block 640 pads the clusters with unexecuted nodes by adding basic blocks that were not executed during the profiling to clusters that have capacity, i.e. where the sum of the weights of the nodes in the cluster is less than N_i and where adding the un-executed basic block will not cause the sum of the cluster's weights to exceed N_i . This padding is performed to reduce unused memory. If there are still basic blocks remaining that have not been added to any of the clusters, and if there is no more cluster capacity to absorb these basic blocks, then process block 640 groups these unexecuted nodes into additional clusters of their own. The technique of adding the un-executed basic block nodes to the clusters or of putting them into their own clusters is called bin packing. A greedy bin packing heuristic, which is well known to those skilled in art such as the teachings of Garey et al. (Garey, Michael and Johnson, David S. *Computers and Intractability, A Guide to the Theory of NP Completeness*, pages 124-127, W. H. Freeman &

Co., San Francisco, Ca., 1979) may be used to perform the bin backing. Process block 645 then creates the next level PEG G_2 from the level 1 partitioned PEG $Chust(G_1, N_1)$. In this graph $G_2 = (V_2, E_2)$ constructed by process block 645, each node in V_2 corresponds to a cluster in G_1 , and each node in V_2 has a weight of one. This weight corresponds to the assignment of N_1 bytes to the basic blocks in each cluster. An edge $e = (u, v)$ exists in E_2 if and only if both u and v are in V_2 , u and v correspond to different clusters, and there is some edge in E_1 from one of the nodes in the cluster represented by u to some node in the cluster represented by v , or visa versa. The weight of e is the sum of the weights of all edges in E_1 that have endpoints in the two clusters (u and v).

Higher levels of the memory hierarchy, i greater than 1, are processed by a loop comprising decision block 650 and process blocks 655, 660, 665, and 670. Decision block 650 begins the loop for each level i of the memory hierarchy. As the loop begins, decision block 650 determines if level i is less than or equal to a maximum number of levels. If so, then process block 655 increments level i , and then process block 660 sets n_i equal to the size of the memory hierarchy for level i , i.e., to the page size in bytes for the page level of the memory hierarchy. Thereafter, process block 665 applies a heuristic H to partition G_i into clusters having sizes no greater than $N_i = n_i/N_{i-1}$ to produce $Chust(G_i, n_i/N_{i-1})$. Process block 665 may perform this clustering in the same manner as process block 635 or it may use a different partitioning heuristic. Thereafter, process block 670 creates the next level PEG G_{i+1} from the level i partitioned PEG $Chust(G_i, n_i/N_{i-1})$. In this graph $G_{i+1} = (V_{i+1}, E_{i+1})$ constructed by process block 670, each node in V_{i+1} corresponds to a cluster in G_i , and each node in V_{i+1} has a weight of one. An edge $e = (u, v)$ exists in E_{i+1} if and only if both u and v are in V_{i+1} , u and v correspond to different clusters, and there is some edge in E_i from one of the nodes in the cluster represented by u to some node in the cluster represented by v , or visa versa. The weight of e is the sum of the weights of all edges in E_i that have endpoints in the two clusters (u and v). Process block 670 performs this PEG construction in the same manner as process block 645. Thereafter, processing loops back to decision block 650, the beginning of the loop, to process the next level $i+1$.

Returning now to decision block 650, if level i is not less than or equal to a maximum number of levels, then process block 675 reorders basic blocks in memory by grouping all of the nodes of a

1 cluster in an adjacent order; process block 680 adds NOPs to fill out the unused portion of memory
2 for that memory hierarchy level for each level of the memory hierarchy; process block 685 stores the
3 reordered and padded basic blocks in memory beginning with the first memory location after a
4 boundary for all the memory levels; and then the process ends at process block 690.

5
6 Referring now to Figure 7, an expansion of process block 670 is illustrated. After process
7 block 665 partitions the PEG, then process block 710, process block 720, process block 730, and
8 process block 740 construct a Program Execution Graph G_{i+1} for the next higher level $i + 1$ wherein
9 the nodes N_{i+1} of G_{i+1} are constructed from the clusters $Clust(G_i, n_i/N_{i,1})$ of G_i . Process block 720
10 assigns a weight of one to each node in N_{i+1} . Process block 730 assigns edges E_{i+1} in G_{i+1} by placing
11 an edge (u,v) between u and v , for each u and v of N_{i+1} , if and only if there is an edge between a pair
12 of nodes in G_i , one of which is in the cluster corresponding to u and one of which is in cluster
13 corresponding to v . Process block 740 assigns weights to the edges E_{i+1} in G_{i+1} ; the weight of an
14 edge (u,v) in G_{i+1} is the sum of the weights of the edges between component nodes in G_i of the
15 clusters, corresponding to u and v . Thereafter, processing loops back to decision block 650, the
16 beginning of the loop, to process the next level $i+1$.

17
18 In an alternative embodiment of the present invention, only a single loop may be used for
19 processing all levels of the memory hierarchy. In this alternative embodiment, process block 665 may
20 be modified to include the functionality of process block 635 so that process block 665 partitions all
21 PEG's. Process block 670 may also be modified to include the functionality of process block 645 so
22 that it creates the next level PEG from all partitioned PEG's. Process block 640 may be moved within
23 the loop to pad the first or additional level clusters.

24
25 After the clustering is completed, the basic blocks are reordered in memory by grouping all of
26 the nodes of a cluster in an adjacent order (process block 675), although the ordering of the basic
27 blocks within this adjacent grouping may be varied. Because clusters must not cross boundaries of
28 memory hierarchies, NOPs are added to fill out the portion of a memory hierarchy level that is not
29 filled by the clusters. For memory hierarchy level 1, NOPs are added to the end of any clusters that

1 have size less than some multiple of the cache line size so that the amount of memory required for
2 each cluster is some integral multiple of the cache line size. For each subsequent level of the memory
3 hierarchy, the minimum number of NOPs necessary to fill out the unused portion of memory for that
4 memory hierarchy level are added (process block 680). After the processing for the last memory
5 hierarchy is completed yielding the final PEG, the reordered basic blocks are stored in memory
6 starting at a boundary of all the memory levels so that all of the basic blocks from one cluster are
7 adjacent, i.e. interleaving from different clusters is not allowed (process block 685). This is true for
8 every level of the hierarchy.

9
10 Referring now to **Figure 8**, a block diagram illustrates a computer system 800 used in
11 performing the method of the present invention, forming part of the apparatus of the present
12 invention, and which may use the article of manufacture comprising a computer-readable storage
13 medium having a computer program embodied in said medium which may cause the computer system
14 to practice the present invention. The computer system 800 includes a processor 802, which includes
15 a central processing unit (CPU) 804, a memory 806, and a multi-level memory cache 810 (level 1
16 cache (L1) 812 through level n cache (Ln) 814). Additional memory, in the form of a hard disk file
17 storage 816 and a computer-readable storage device 818, is connected to the processor 802.
18 Computer-readable storage device 818 receives a computer-readable storage medium 820 having a
19 computer program embodied in said medium which may cause the computer system to implement the
20 present invention in the computer system 800. The computer system 800 includes user interface
21 hardware, including a mouse 822 and a keyboard 824 for allowing user input to the processor 802
22 and a display 826 for presenting visual data to the user. The computer system may also include a
23 printer 828.

24
25 Although the present invention has been particularly shown and described with reference to a
26 preferred embodiment, it will be understood by those skilled in the art that various changes in form
27 and detail may be made without departing from the spirit and the scope of the invention.